A Pilot Study of the Safety and Usability of the Obsidian Blockchain Programming Language

Gauri Kambhatla
Electrical Engineering & Computer Science, University of Michigan, Ann Arbor, MI, US

Michael Coblenz
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Reed Oei
Department of Computer Science, University of Illinois, Urbana, IL, USA

Joshua Sunshine
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Jonathan Aldrich
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Brad A. Myers
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Although blockchains have been proposed for building systems that execute critical transactions, security vulnerabilities have plagued programs that are deployed on blockchain systems. The programming language Obsidian was developed with the purpose of statically preventing some of the more common of these security risks, specifically the loss of resources and improper manipulation of objects. The question then is whether Obsidian's novel features impact the usability of the language. In this paper, we begin to evaluate Obsidian with respect to usability, and develop materials for a quantitative user study through a sequence of pilot studies. Specifically, our goal was to assess a) potential usability problems of Obsidian, b) the effectiveness of a tutorial for participants to learn the language, and c) the design of programming tasks to evaluate performance using the language. Our preliminary results tentatively suggest that the complexity of Obsidian's features do not hinder usability, although these results will be validated in the quantitative study. We also observed the following factors as being important in a given programmer's ability to learn Obsidian: a) integrating very frequent opportunities for practice of the material – e.g., after less than a page of material at a time, and b) previous programming experience and self-efficacy.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Human-centered computing → User studies; Human-centered computing → Usability testing

Keywords and phrases smart contracts, programming language user study, language usability

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.2

1 Introduction

Blockchains have a myriad of applications, including shipping, supply chain, auctions, storing health records, and voting [8]. A blockchain is used when a central authority cannot be trusted; instead of a singular ledger (as is used by an entity like the Federal Reserve), a distributed ledger is used to keep track of transactions that are made, held accountable by the users of the blockchain themselves. Smart contracts are programs that are deployed across a blockchain network. Since blockchain technology is often used in potentially high-stakes contexts, such as financial transactions, if a bug in a contract is exploited, it could involve loss of important resources (like money or personal items). Some of the more common of these security risks are (1) loss of resources and (2) manipulating objects at improper times. Current status quo blockchain languages (such as Solidity) have no mechanism to prevent such bugs.

Obsidian is a programming language for writing smart contracts that is designed to prevent these two issues; it uses both typestate and linear types to prevent such risks statically [2]. In order to detect and prevent these vulnerabilities, Obsidian introduces the concepts of ownership, assets, and states. Ownership is a property of *references*, rather than the objects themselves. There are three types of references; `Owned`, `Unowned`, and `Shared`. An object can have exactly one `Owned` reference, and any number of `Unowned` references. It can have any number of `Shared` references, but only if there is no `Owned` reference. Making an object an *asset* enables the compiler to prevent losing track of `Owned` references. Assets, combined with permissions (the types of ownership references) allow the compiler to detect the loss-of-resource security bug. Objects can also have *states*, and certain transactions can only occur within a particular state. States help prevent improper manipulation of objects; certain contracts might allow certain transactions only when in an appropriate state. A *contract* is similar to a class in other object-oriented languages, with corresponding *transactions* (analogous to methods) and fields (like member variables) [2].

Yet how do we know Obsidian is effective in accomplishing what it is designed to achieve? Do users actually make fewer of these security mistakes in Obsidian than in other common smart contract programming languages, such as Solidity? And how usable is Obsidian in the long term, after programmers have mastered the language? In this paper, we focus on beginning to answer these questions; we want to see whether the bugs that Obsidian detects are ones we observe people inserting frequently in the laboratory. In addition, we want to see whether the language is in fact usable, and how we can teach it to potential users.

We intend to run a user study where participants will learn either Solidity or Obsidian through a tutorial, and then attempt a series of programming tasks that will test their knowledge of the language. We will also assess the safety of the code they have will write. In order to develop these programming tasks and the Obsidian tutorial, we first ran a series of pilot studies. This paper describes the design and results of those pilot studies, addressing in particular, the following research questions:

RQ1 What are the most significant usability problems with Obsidian?

RQ2 How should a tutorial be best authored to help one learn a language sufficiently well to get to the point where usability can be evaluated?

RQ3 How do we design programming tasks that assess the ability to program effectively and also potentially expose the types of bugs that our tool would detect or prevent?

The answers to these research questions, and what we learned from this study could also be of interest to a broader audience; specifically, the initial findings we report in this paper that could feed into future work are as follows:

- A potential design of a method of teaching programmers to effectively learn an entirely new language composed of complex features
- Hypotheses on why some programmers are more effectively able to learn and utilize a new programming language than others
- Evidence that the additional features Obsidian introduces for security can actually be effectively used by programmers

It is important to note the findings we present here through our pilot studies are not fully validated; we plan to substantiate them in our follow-up quantitative user study.

2 Related Work

Designing programming tools and languages has been shown to be better when done with a human-centered approach [4]. It is often inadequate to just use technical methods in assessing properties of a language. For example, even if type soundness has been proven,

the language might be too complex for a programmer to learn and use [3]. Programming languages are interfaces between a human programmer and a computer to enable people to effectively create programs, and as a result, should be tested and evaluated through user studies with actual developers. Obsidian has been designed using such a human-centered approach [1], and is similarly being evaluated through user studies.

There are differences in *approaches* to programming between novices and experts [6]. Novices generally lack detailed mental models of what they are doing, and approach programming one line at a time, rather than as meaningful blocks [6]. On the other hand, experts tend to be faster and more accurate, able to make use of a variety of effective strategies and more vast knowledge base [6]. The participants in the pilot studies fall somewhere between novice and expert, and their approach to programming may play a role in their success in the tasks given to them. While knowledge is the ability to state how something works, strategies are the ability to apply that knowledge. There are many different strategies programmers use to comprehend programs, including systematic (tracing through all the code in detail), as-needed (only looking at code related to the required task), and inquiry episodes (asking a question, hypothesizing an answer, and verifying by looking at code, or compiling the program), among others [7]. The programming strategy participants in our study use is a variable that could influence how they learn Obsidian.

In addition, studies of novice programmers have shown that past experience is strongly predictive of self-efficacy, and that a strong mental model increases self-efficacy [5]. Ramalingam et al. also found that both of these (self-efficacy and strong mental models) affect performance in an introductory class setting. Active learning helps develop these mental models; it requires an individual to be active in constructing their own knowledge and finding new ideas [9]. An active learning environment is beneficial to learning programming because it is one that provides immediate feedback and engaged participation, which can help develop mental constructions of programming concepts [9]. Although Obsidian is not being taught to novice programmers in a course setting, self-efficacy and mental models might still play roles in the ability to learn the language effectively, and teaching through active learning might help participants pick up the language more successfully.

3 Study Design

In this work, we focused on designing the Obsidian condition for the quantitative user study. Participants in our pilot studies were asked to do the Obsidian tutorial, and then some more in-depth programming tasks, during which they were asked to think out loud. They were told they could ask questions about anything they didn't understand throughout the study; the answers to these questions could then be incorporated in the next iteration of the tutorial and tasks. This pilot study was approved by our IRB, and participants were paid $10/hour for their time during the study.

We chose tasks with potential real blockchain use-cases for external validity; by approximating code an actual smart contract programmer might write, we hope to be able to generalize our conclusions to our proposed end applications of Obsidian. Each task had a series of parts that tested different aspects of Obsidian, and are described below, along with the design of the tutorial.

Obsidian Tutorial

The tutorial consisted of a Qualtrics survey with a few multiple choice, write-in answers, short answer questions, and small programming questions. It was split into eight sections: four on ownership, one on assets, and three on states. An early version of the tutorial had only four

sections, but we subdivided it because of participant feedback of having too much information at once with just a few sections. Participants completed programming questions in a separate Visual Studio Code (VSCode) window, in which participants could compile their code to confirm that they had complied with the type system's requirements. Some of the short-answer questions were purposefully open-ended, as in "Describe the relationship between ownership and states in your own words." The objective was to make the participant think about, and thereby internalize, these concepts, rather than to evaluate their understanding.

Auction Task

The **Auction** task simulates an English auction; there are multiple Bidders who each make a Bid for a single Item being sold by a Seller. The highest Bidder receives the Item for the price of the highest Bid. However, unlike in a normal English auction, when a Bidder makes a Bid, they give the Money to the Auction house *immediately*, and the Money is returned to that Bidder if another Bidder makes a higher Bid. This requirement ensures that all bids are legitimate, rather than allowing for a failure mode in which the sale cannot be completed because a bidder does not pay for the item. A consequence is that the contract must return the bid money every time a higher bid is made. If a participant forgot to do this, they would lose a resource (the bid money), which the Obsidian compiler would detect.

The task included five parts. The most significant parts of this task were 3 and 4, which allowed for a potential loss-of-resource bug. **Part 3:** Write code to return money to a Bidder in the case that the bid is not greater than the current maximum bid. This part (whose starter code is shown in Listing 1) was created to prime the participant for the next question, so the participant is aware there is a way to return money to a Bidder. **Part 4:** Update the current maximum bid (i.e., given that the new bid is greater than the previous, replace the `maxBid` with the new one). Since Money is an asset, and a Bid contains Money, the participant will not be able to simply do something like `maxBid = newBid`; the money will have to be returned to the Bidder before overwriting `maxBid`, otherwise they will get a compiler error in regards to overwriting an asset (losing a resource). This part (starter code also shown in Listing 1) was meant to capture this potential pitfall. To implement this correctly, a participant must return the Money to the Bidder and disown the Bid. They could write code to do this or call a given function that does both these things.

Listing 1 Partial Starter Code for Auction Parts 3 & 4.

```
transaction makeBid(Auction @ Open this, Bid @ Owned >> Unowned newBid,
                    Bidder @ Unowned bidder) {
   if (newBid.getAmount() > bid.getAmount()) {
      setCurrentBid(newBid);
      Bidder tempBidder = maxBidder;
      maxBidder = bidder;
   }
   else {
      //Part 3. TODO: return the newBid money to the bidder.
      //You may call any other transactions as needed.
   }
}
transaction setCurrentBid(Auction @ Open this, Bid @ Owned >> Unowned b) {
   //Part 4. TODO: set the current bid to the new bid b.
   //You may call any other transactions as needed.
}
```

Pharmacy Task

The **Pharmacy** task simulates a pharmacy; there are Prescriptions, PrescriptionRecords (which keep track of Prescriptions), and Patients (who have Prescriptions), as well as the Pharmacy contract, which has a list of PrescriptionRecords. The goal of this task is to have the participants utilize states to prevent improper use of the prescription; a Patient can only fill a Prescription when they have additional refills. There were three parts to this task; the third was most significant, which had a participant writing in lines of code to fill a prescription. In this part (starter code shown in Listing 2), the participant must take an element off a list, apply transactions to that object, and add it back to the list. It assesses understanding of the code, and the ability to use the language to implement what the participant wants to occur, as well as correct use of states.

Listing 2 Partial Starter Code for Pharmacy Part 3.

```
transaction fillPrescription(Prescription @ Unowned prescription) {
   MaybeRecord maybeRecord = prescriptionList.removeIfExists(prescription);
   // TODO Part 3: Fill in the rest of this transaction.
   // You will need to call the doFill transaction in this
   // class (Pharmacy) on the appropriate PharmacyPrescriptionRecord.
   // Be sure to record that the prescription has been filled.
}
```

Gambling Tasks

The **Gambling** task simulates betting at a Casino; before every Game, Bettors place a bet on the outcome of the Game. A set of restrictions and assumptions is given to the participant, as well as a sequence diagram showing a potential timeline of possible events, and structural diagram explaining how contracts relate to each other. This task is purposefully more open-ended; the participant can design and implement the Casino contract however they would like using Obsidian, but the program must comply with the given requirements. The goal of this task was to see how the participants used what they learned about the language to design their own program (and be able to implement it). Code for the other contracts (Bettor, Money, etc.) is not shown here due to space constraints.

Listing 3 Partial Gambling Starter Code.

```
main asset contract Casino {
    Money @ Owned money;
    Game @ Owned currentGame; //The game that is currently being played
    BetList @ Shared bets; //The bets for the current game being played

    Casino @ Owned() {
        money = new Money(100000);
        currentGame = new Game();
        bets = new BetList();
    }
    //TODO: Add your code here.
}
```

4 Initial Results

Since these studies were exploratory, serving as preparation for our evaluative user study, we changed the tutorial and tasks after nearly every participant, based on the qualitative results and participant feedback. We recruited six participants (three men, and three women),

all of whom were undergraduates in computer science at different universities. Despite all the participants being almost the same amount through their undergraduate career (they were either rising Juniors or Seniors), there was a wide range of 3 - 9 years of previous programming experience, and they all had a different CS education. All the participants were familiar with Java to varying degrees (ranging from 1.5 to 8 years of experience), which was a requirement for participation, since Obsidian is similar to Java. None of the participants had any previous experience with blockchain programming. Each participant was given an anonymous participant ID: R0, R1, etc.

Participants worked on a 15" MacBook Pro with a second monitor that displayed the documentation pages and instructions for the tasks. We recorded screen and audio of each session. The programming tasks were all done in VSCode, for which we created a plugin for Obsidian syntax highlighting, and through which we ran the Obsidian compiler.

Some participants did not seem to understand and internalize the information given in the tutorial despite reading it. For example, R2 said the idea of ownership made sense theoretically, but not in application. This may be because R2 missed some key concepts, like the fact that ownership is a property of a reference, not an object. In contrast, other participants seemed to fully comprehend the material; R1, R3, and R5 got nearly all questions correct. A summary of the tutorial results are shown in Table 1.

Table 1 Tutorial Results.

Participant	Tutorial Version	Questions Correct	Time
R0	2 Parts, No questions	N/A	N/A
R1	4 Parts, all multiple-choice	19/19	35 min
R2	4 Parts, all multiple-choice	11/19	40 min
R3	8 Parts: multiple-choice, short answer, programming	20/22 (non-code), 8/8 (code)	1.25 hours
R4	8 Parts: multiple-choice, short answer, programming	14/22 (non-code), 4/8 (code)	2.5 hours
R5	8 Parts: multiple-choice, short answer, programming	20/22 (non-code), 8/8 (code)	53 min

All the participants except R4 did the **Auction** task. They all had the most trouble with part 4; neither R0 nor R2 was able to complete it. Every participant who did part 4 started by writing `bid = newBid`, which overwrote an owned reference to an asset; the compiler generated an appropriate error message. Participants R1, R3, and R5 realized that ownership of the original bid must be transferred first. R1 even said out loud after typing this in, "Oh is bid an asset? Yes, it is an asset. Then this should fail." After compiling the code and confirming the failure, R1 made sure the original bid was transferred. R3 did the same, but first returned the Money in the Bid to its Bidder. R5 was the only one that used the given transaction `returnBidMoney()` (which gave the Money back to the Bidder and disowned the current bid).

All the participants except R4 were given the **Pharmacy** task. R0 was able to complete parts 1 and 2, but was unable to do any of the third part; the participant could not figure out what to do, and was stopped by the experimenter due to time constraints. Participant R2 did part 1 incorrectly, and part 2 correctly. R2 was unable to figure out the third part, and stopped after a significant amount of trial and error due to time constraints. R2 was asked by the experimenter to answer in pseudocode; the participant did this mostly correctly

(but forgot to check if the PrescriptionRecord actually existed). Participant R3 did the whole task correctly, but forgot to append the PrescriptionRecord back on the list after consuming a refill. Participants R1 and R5 did the entire task correctly.

Only R3 and R5 were given the **Gambling** task (it was created after R1, and both R2 and R4 could not take it because of time constraints). R3 took 1 hour to complete the task and get the code to compile. Most of this time was spent on understanding the architecture of the objects given, and the structure of the problem itself. Participant R5 took 36 minutes to do this task and get the code to compile. In this version, the requirements were unchanged, but the problem was made more clear, and there were fewer layers of abstraction. For R5, an architectural diagram was added to show how objects are related (e.g., a Bet has a Bettor and a BetPrediction), in addition to the sequence diagram (showing an example of potential actions) given to R3. Both R3 and R5 successfully designed and implemented a program that met all the requirements given in the specification. They made correct use of states and permissions. R3's program was 63 lines long, and R5's was 56 lines long.

5 Discussion

A clear distinction can be drawn between the results of the different participants. Participants R0, R2, and R4 struggled with the study (both the tutorial and the programming tasks), while participants R1, R3, and R5 found the tutorial and tasks easy to understand. There was almost no middle ground; one of the challenges in designing the tutorial and tasks was that after one participant, it seemed the exercises were very difficult and needed to be simplified, but after another, they seemed too simple. While R0, R2, and R4 needed prompting and additional explanation, and still did not finish all the exercises, R1, R3, and R5 easily completed the tutorial (including the programming exercises for R3 and R5). These three participants were able to work through the programming tasks, write code in the language, understand the compiler errors, and make fixes when necessary. They had understood the new *concepts* they were taught; while working through the Pharmacy task, R5 said "... and this takes an owned [reference] and makes it unowned...," showing that the participant grasped the idea of ownership. Both R1 and R3 also said similar things while thinking out loud. On the other hand, R0, R2, and R4 got stuck; it was clear that although they read through the tutorial, they did not fully understand the concepts. For example, despite reading in the documents, answering questions about it in previous parts of the tutorial, and given an explanation by the experimenter, R4 did not seem to understand how state and permission transitions worked as types for a parameter in a transaction declaration.

Below are hypothesized explanations for the differences among participant performance:

- **General programming experience**. The participants who struggled had an average of 3.2 years of programming experience; the others had an average of 7.5 years of experience. One of the key observations we made during the pilot studies was that the participants who struggled had the most trouble because upon getting an error, they would either get stuck or start trying random solutions that neither made sense conceptually nor syntactically. Participants who successfully completed the tutorial and tasks used a variety of techniques to fix compiler errors. They had less trouble understanding the compiler messages (most of the time, they knew what the problem was immediately), and even if they did not, they used strategies to find a solution. These observations suggest that the amount of past programming experience played a role in how effective participants were in using the unfamiliar language.

- **Object-oriented (OO) programming experience**. The less effective participants had an average of 2.2 years of Java experience, and the more effective ones had an average of 5.3 years (familiarity with Java was a prerequisite to participating in this study). Perhaps some concepts that are learned in OO classes, or learned through developing in OO languages affects the way one learns other OO languages.
- **Teaching style**. The tutorial was text- and exercise-based. Perhaps the participants who struggled may have done better if the material had been taught in a video, or lecture format, or in a more interactive way.
- **Type of programmer**. Different programmers use different programming strategies (see §2). Perhaps some of these strategies are more effective in learning or using Obsidian than others. While there were systematic and opportunistic strategies used in the group that was more effective, the ones who had more trouble only used an opportunistic approach.
- **Self-efficacy and interest**. Self-efficacy appeared to play a role in how effective participants were in using the language while completing the tasks. The effective participants were very confident; they were not afraid to question the experimenter about things they did not understand and things they thought were wrong in the tutorial or tasks. They were also more relaxed; the participants who struggled seemed tense. It is difficult to tell whether confidence entailed being effective in completing the tasks, or whether doing well made participants more confident. In addition, lower self-efficacy might have played a role in whether participants asked questions when they were confused, and thus how well they performed. The effective participants also had a genuine *interest* in what they were learning. They made noises of surprise or interest while reading the tutorial, saying for example "Oh, that makes sense", or "huh, that's interesting." In contrast, other participants went through the study like taking an exam, or doing an assignment; they were just doing exercises they were given. Perhaps having an interest in learning a new language made participants more successful at completing the given exercises, or at least created a more open mindset that might have allowed for faster debugging.

RQ1 asked about identifying potential usability problems with Obsidian. Multiple participants mentioned things related to the environment; a few expected more precise autocomplete (VSCode has a default autocomplete that lists any words used previously in the window, which confused some participants), and one asked about in-place error diagnostics (showing errors as one types). In addition, a few participants were confused by some of the compiler errors. Two participants tried to add a permission or state to an object they were creating, like `g = new Game@FinishedPlaying()`. The error message was `Error: '(' expected, but @ found`, which they eventually figured out. R2 did not understand the compiler message `variable is an owning reference to an asset, so it cannot be overwritten` at first, and when it was explained by the experimenter, was unable to figure out how to resolve the problem. R5 was confused by the error message `Can't reassign to variables that are formal parameters or which are used in a dynamic state check.` R5 commented that even distinguishing between the two (whether a formal parameter or used in a dynamic state check) might be more helpful. None of the participants had much feedback about the language itself, although there was a general consensus that states were an easier concept to grasp than ownership because they are more familiar.

RQ2, asked about how to design a language tutorial. Teaching any new programming language is hard; teaching a new programming language that has more complex features may be even harder. From our results, we learned a few things: (1) material that should be read needs to be split into manageable sections, (2) simply giving people material to read is

not enough for them to understand it – they need to be given questions along the way to force them to thoroughly comprehend it, and (3) these questions must have them actually *do* problems themselves, rather than merely pick an answer choice. When given large amounts of reading material at a time, participants had trouble remembering all the information; an early participant commented that they wanted to see an example when doing the programming tasks, and while they knew there was probably one in the documentation, they had no idea where to look. After splitting the reading material into eight sections (each less than a page), we observed that later participants found it easier to understand the material initially, but that this also enabled going back to particular documents for reference when doing the programming exercises and tasks. Questions (in particular, questions that have someone *write* or *do* or *try* something) seemed more effective than only having participants read text. Participants who used the version of the tutorial with programming questions said they were helpful, and one participant who did not specifically stated that being able to try out the code would have helped learn and internalize the material more.

RQ3 asked about designing programming tasks that:

1. assess the ability to program effectively;
2. expose the types of bugs likely to be made by participants in the control condition

The Auction task was mostly straightforward except for part 4, in which everyone who attempted it made the same mistake of overwriting an asset. This would have caused the loss of a resource (money) if it had not been caught by the Obsidian compiler. This does seem to be a relatively common security bug, since every participant made the mistake at first. It therefore seems likely that Solidity participants in the user study would insert the same bug as well, but would not have the compiler to remind them of the error. This suggests that this task fulfills goal (2). The Pharmacy task required an understanding of how contracts interact with each other, as well as using states to prevent improper manipulation of objects. The task was intended to require higher-level insight from participants, so the instructions for the third part in this task do not explicitly lay out everything the participant needs to do. The task requires the participant to be able to understand Obsidian code (including the novel concepts of ownership and states). In the Gambling task (again only given to R3 and R5), the participants wrote 50+ lines of code. Though a small program, this is not a trivial amount of code, especially since they were able to design a program to follow the specifications accurately, implement their design correctly using concepts they had only just learned (ownership and states) and have their code compile using syntax that was new to them for this study. As a result, the Pharmacy and Gambling tasks seem to fulfill (1).

In this pilot study, we started to gather evidence that the new language features of Obsidian that allow for writing safer smart contracts can be used effectively by real users. The Auction task had participants use ownership to prevent the loss of a resource, and the Pharmacy task had participants use states to prevent incorrect manipulation of objects. All the participants who did these tasks were able to use ownership and states, and all of the more experienced ones used ownership and states to complete them correctly. Although the features of the language that allow for greater safety are more complicated, and one might assume make the language less usable, we did not observe this in these initial results. This will need to be more fully validated with additional participants in our user study.

6 Threats to Validity

The participants were a convenience sample of undergraduates studying computer science; this may not be representative of the general population of programmers. In the quantitative study, we hope to have a more diverse group of participants who more closely align with the

intended users of Obsidian. We only had six participants, and not all of them attempted all parts of the study. In addition, some of our participants had limited object-oriented programming experience. As a pilot study, our goal was to refine our study design and hypotheses, and for those purposes, we believe our approach sufficed.

7 Conclusion and Future Work

We designed and conducted pilot studies to find potential usability problems with Obsidian, its tutorial, and the tasks used to evaluate it. We created a tutorial that helps at least some participants learn Obsidian well enough to use it effectively. We also created programming tasks that test a participant's ability to use Obsidian competently and have the potential for participants to make the types of security bugs that are caught by Obsidian's compiler. We identified hypotheses for why some of our participants found it easier to learn and use Obsidian than others, and more generally, why picking up new languages is easier for some people than it is for others. We found that an incremental approach of teaching the language that included regular practice opportunities was most effective, but that participants with more programming experience appeared to be much more successful at completing the tasks.

In the future, we will conduct a quantitative user study to compare Obsidian to a control language with respect to usability and security, and will continue to make any necessary changes to our tutorial and programming tasks. In the process, we hope to evaluate our hypotheses regarding why some participants are more successful in learning the language and completing the programming tasks, with a focus on self-efficacy and experience.

References

1 Celeste Barnaby, Michael Coblenz, Tyler Etzel, Eliezer Kanal, Joshua Sunshine, Brad Myers, and Jonathan Aldrich. A User Study to Inform the Design of the Obsidian Blockchain DSL, 2017.

2 Michael Coblenz. Obsidian: A Safer Blockchain Programming Language. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 97–99, Piscataway, NJ, USA, 2017. IEEE Press. event-place: Buenos Aires, Argentina. doi:10.1109/ICSE-C.2017.150.

3 Stefan Hanenberg. Faith, Hope, and Love: An Essay on Software Science's Neglect of Human Factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 933–946, New York, NY, USA, 2010. ACM. event-place: Reno/Tahoe, Nevada, USA. doi:10.1145/1869459.1869536.

4 B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer*, 49(7):44–52, July 2016. doi:10.1109/MC.2016.200.

5 Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. Self-efficacy and mental models in learning to program. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '04*, page 171, Leeds, United Kingdom, 2004. ACM Press. doi:10.1145/1007996.1008042.

6 Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, June 2003. doi:10.1076/csed.13.2.137.14200.

7 M. A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2):183–207, March 2000. doi:10.1016/S0167-6423(99)00036-2.

8 Dmitrii Suvorov and Vladimir Ulyantsev. Smart Contract Design Meets State Machine Synthesis: Case Studies. *arXiv:1906.02906 [cs]*, June 2019. arXiv: 1906.02906. URL: http://arxiv.org/abs/1906.02906.

9 Edward Zimudzi. Active learning for problem solving in programming in a computer studies method course. *Educational Sciences*, 3(2):9, 2012. URL: https://ubrisa.ub.bw/handle/10311/1170.